OAuthLite from CourseSmart

CourseSmart and their business partners exchange a variety of web service invocations in one direction or another—either a partner requesting a service from CourseSmart or vice versa. It’s important that these exchanges be conducted with a high degree of integrity and privacy. CourseSmart has adopted OAuth has its preferred method of implementing secure web service exchanges. OAuth is documented at <http://www.oauth.org>. The Wikipedia also has a useful summary of OAuth.

OAuthLite is a proper subset of OAuth that’s useful for conducting these exchanges.

## Why is it called ‘Lite’?

OAuthLite is a ligher version of OAuth in the following sense:

* OAuthLite is “two-legged”  
  Full OAuth supports three primary roles User, Consumer and Provider. This “three-legged” authentication requires multiple flows and is primarily intended to implement use cases where one service provider (Provider) delegates temporary authority to another service provider (Consumer) to a User’s assets.  
  Typical web service scenarios only have two roles: Service Provider and Service Consumer. OAuthLite only implements enough of the protocol to support a secure request from the Consumer to the Provider. This OAuth scenario is often called “two-legged” OAuth and it’s corresponding use case is often termed either “Signed Request” or “Phone Home”.
* OAuthLite only supports a single signing protocol  
  Full OAuth allows multiple digital signing protocols to be supported and negotiates the signing convention with its partner. OAuthLite only supports HMAC-SHA1 as its signing protocol. However this protocol is universally implemented in all languages, is widely supported, and is in reality the protocol used by virtually all OAuth exchanges, Lite or otherwise.

# OAuthLite package contents

The OAuthLite package is a zip file that contains:

1. The OAuthLite API  
   This is the low-level API that either invokes an OAuthLite signed request to a service, or it validates an incoming OAuthLite signed request from a consumer. In other words, it performs the OAuthLite treatment from either the sender or the receiver’s point of view.  
   This API will be used by you if you’re integrating OAuthLite signing into your own program.
2. The OAuthLiteServlet  
   This is a higher-level component that is a complete ready-to-deploy servlet for receiving OAuthLite requests. This servlet comes prepackaged with a few test services. You can then add to these service or modify the sample services to enable the OAuthLiteServlet to serve up your own OAuthLite compatible services.
3. Source code for OAuthLite  
   All source code for the OAuthLite libraries, servlet, and companion services is contained herein. This is useful if you wish to integrate these features more deeply into your own infrastructure, or just want to know how it works.

The source code is in multiple languages with more to come on an as-needs basis. The current languages included are Java and Python.

### Dependencies for Java OAuthLite

The dependent jars for OAuthLite are included in the /server/WEB-INF/lib directory. If you wish to use your own versions or integrate with a maven-style build you’ll need:

* Apache HttpClient and its dependencies
* Google JSON (Gson) jar
* www.oauth.net oauth.jar

### Dependencies for Python OAuthLite

Python OAuthLite requires the Python oauth egg available at [www.oauth.net](http://www.oauth.net). The zip file that contains a easy\_setup is included in this zip in the /components directory.

# OAuthLite API

OAuthLite implements only two API calls between partners—one for making a secure service request and the other for verifying the integrity of a received request.

To ensure the security of the exchange CourseSmart and its partner must share a “secret” between them that will be use to digitally sign the API call. This secret has two values associated with it that are labeled <key> and <secret> parameter. These two values need to passed into each of the API calls below. CourseSmart will generate these values and provide them to you via an out-of-band information exchange (typically through a phone call or pair of emails).

The <key> gives a logical and public name to the shared secret. It is used by CourseSmart’s multi-tenant hub to look up the secret value. The <secret> is the secret value itself.

### Send an OAuth signed service request to a partner provider

resultMap = postSignedRequest( url, paramMap, key, secret)

This call uses an Http POST to send a signed request. It returns a dictionary/map type of object (depending on language binding).

The <url> argument is the full URL that points to the exact partner service. For example, to invoke the CourseSmart ServiceHub you might use a url such as:

[http://nimbus.coursesmart.com/ServiceHub/ViewETextBook](http://nimbus.coursesmart.com/ServiceHub/ViewDesktop)

CourseSmart will give you the syntax of the url to use.

The <paramMap> argument is a dictionary/map of name-value pairs that are used to parameterize the service.

For example, a consumer might call the GetETextBookLink with the paramMap: {‘isbn’: ‘12344311133’, ‘page’: ‘47’}. This would cause the service to return a an eTextBook of the specified ISBN and open to page 47.

Once the postSignedRequest is called it will sign the request with a digital signature that inspects every element of the API call (url plus all arguments). It will automatically add a variety of elements to the transmission and then it post it to the partner.

The result of this request is returned as a dictionary/map type of object. This dictionary is either populated with the name/value pairs supplied by the service. If the service failed for some reason, the returned dictionary contains a map with two values:

1. outh\_error\_code  
   The name of the error that has occurred within the service
2. oauth\_error\_message  
   A description of the error that has occurred within the service

### Verify the integrity of a received request from a partner consumer

resultMap = verifySignedRequest( httpRequest, key, secret)

The receiver of a partner signed transmission needs to reverse the signature process that the preceding API call encoded. Since this message has been POSTed to this receiver it is assumed that that receiver has “caught” the message in some Http server such as Apache or Tomcat. These servers put the full context of all received messages in a HttpRequest object (familiar to any web programmer).

The verifySignedRequest inspects the full message and the digital signature. Assuming that it fully checks out, this method will strip out all of the OAuth headers and return a resultMap that is the duplicate of the map passed into the postSignedRequest.

If the signature fails in any way an Exception will be thrown by this method. Is the responsible of the message handler to use this exception in reasonable way.

The two standard exceptions that can be thrown are:

* Invalid signature

The signature did not match the API values

* Signature timed out  
  The timestamp on the digital signature indicates that the sender’s time was more than five minutes different than the receiver’s time. This can happen through tampering or throug server clock skew (one server’s clock significantly different than the partner’s—(ignoring time zones)).

# OAuthLiteServlet

The OAuthLite API layer described above is suitable for incorporating OAuthLite into your programs. By contrast, the OAuthLiteServlet is a ready-to-deploy Java Servlet that can be dropped right into a Java Web container and run. It comes preconfigured with several services that can be immediately run to test the OAuth connection. For example, the EchoOAuthService echos the input parameters passed into the provider right back to the consumer.

In addition, after you’re satisfied that the OAuthLiteServlet you can then create new services of your own (or modify one of our existing examples) to create a service that you want. This is done by simply creating a new OAuthService class (that implements the IOAuthService interface). Once created, this class is simply dropped into the ‘Service’ folder of the OAuthLiteServlet and that new service becomes available the service consumer.

OAuthLiteServlet is especially useful for CourseSmart partners who need to create services that the CourseSmart ServiceHub accesses. The Examples section below gives two complete examples of these kinds of services.

## Writing a Service for OAuthLiteServlet

1. Inspect the sample services provided in the Service folder of the deployed OAuthLiteServlet.
2. Inspect the service interface that is defined in IOAuthService (also in the Service) directory.
3. Create a new service class. Pick a name for the service (e.g. MyService) and name the service <MyService>OAuthService. This name is important. This convention will allow the OAuthLiteServlet to automatically load the service when it is needed It will also be the string that is appended to the OAuthServlet URL to execute the service.
4. Implement a service that will provide the desired transformation on the Map parameter and will generate the desired outgoing Map. This output map will then be returned to the service consumer.
5. Move the compiled service into the Service folder.

## Testing and Executing a service

1. To test the service alone, with no OAuth or web infrastructure involved,d run the provided TestOAuthService utility from the command-line to test a service. This tester simply displays the results of executing your raw service. It requires command-line arguments of the service name followed by any number of name value pairs.

java TestOAuthService MyService name joe zip 55512

1. Run the provided TestOAuthServlet utility from the command-line to test a service running with all the OAuth protocol and interacting with the OAuthServlet. In addition to the arguments specified above it requires the full url of the servlet path (including service name).

java TestOAuthServlet http//:localhost:8080/OAuthLine/MyService name joe zip 55512

1. Now your ready to test your service with the true partner consumer. If the CourseSmart ServiceHub is your partner consumer, contact consumer to run a test.

# OAuthLite examples

The following examples implement real use cases in which the CourseSmart invokes a partner service.

## Example: ServiceHub needs supplementary user data from a partner institution

### Need:

In certain single-signon (SSO) environments. such as CAS or OpenID, the SSO authenticates and identifies the user but provide no further information about the user. But in the use case that this is the first time a user has come to the CourseSmart service we need additional information; in particular, first name, last name, email address, and role.

### Solution:

The CourseSmart hub invokes an OAuthService on the partner site called GetUserInfo. CourseSmart passes the UserId that’s been provided by the SSO and the GetUserInfo service returns the needed values.

### Recipe:

1. Deploy the OAuthLiteServlet as described above.
2. Before creating your own service inspect the provided OAuthService ‘GetUserInfoService’ in the service folder.
3. Use the TestOAuthService tester described above to try out this service. Enter some phony name as the userId.

java TestOAuthService GetUserInfoService userId fred

1. You should see output from this that contains phony output data based on the userId you provided.

email: fred@example.com

firstname: first\_fred

lastname: last\_fred

roles: Instructor,SysAdmin

1. Now modify the provided service GetUserInfo to add your custom logic.
2. Rerun tester

## Example: Partner wishes to authorize access of certain users to certain ISBNs

### Need:

Some institutions wish to programmatically authorize certains users to certain ISBNs (eTextbook resource ids) based on some institution policy.

### Solution:

Create a custom OAuthService that takes ‘userId’ and ‘isbn’ as input parameters and return a result ‘authorization’ equals ‘true’ or ‘false’

### Recipe:

1. Deploy the OAuthLiteServlet as described above.
2. Before creating your own service inspect the provided OAuthService ‘AuthorizeUserForISBNService’ in the service folder.
3. Use the TestOAuthService tester described above to try out this service. Enter some phony name as the userId.

java TestOAuthService AuthorizeUserForISBNService userId fred isbn 1234432112345

1. You should see output from this that contains phony output data based on the userId you provided.

authorize: true

1. Now modify the provided service AuthorizeUserForResource to add your custom logic.
2. Rerun tester